Laboratorul 4

Shell

1 Variabile

1.1 Definirea si dereferentierea variabilelor

Variabilele shell nu au tip (ex. întreg, caracter, string, vector). Ele sunt inițializate direct umând convenția matematică `variabilă=valoare`.

```
$ a=2009
$ b=alex
$ c="fata merge pe jos"
```

Atenție, nu trebuie să existe spații în jurul semnului `=`.

```
$ a = 2009
ksh: a: not found
```

Pentru a folosi o variabilă, numele acesteia trebuie prefixat cu `$`:

```
$ echo $a, $b, $c
2009, alex, fata merge pe jos
```

La inițializarea unei variabile pot participa alte variabile:

```
$ d="$b: $c ($a)"
$ echo $d
alex: fata merge pe jos (2009)
```

O variabilă este extinsă reinițializând-o cu valoarea veche împreună cu elementele noi:

```
$ name=bond
$ echo $name
bond
$ name="james $name"
$ echo $name
james bond
```

O atentie deosebita trebuie acordata felului in care sunt delimitate variabilele atunci cand sunt folosite. De exemplu, e util sa folosim acolade {} in situatia in care dereferentierea variabilelor creeaza confuzie:

```
$ myname=John
$ echo $myname_Doe

$ echo ${myname}_Doe
John_Doe
$
```

In exemplul de mai sus, shell-ul nu are nici o variabila *myname_Doe* setata ca sa o poata dereferentia, motiv pentru care nu afiseaza nimic (comanda *echo* este responsabila pentru afisarea caracterului *newline*). In schimb, daca intentia era sa dereferentiem valoarea variabilei *myname* si sa construim string-ul *John_Doe*, trebuie folosite {}.

1.2 Tipurile variabilelor

Desi shell-ul nu ia in consideratie tipurile variabilelor (ele sunt de fapt stocate ca string-uri), rutine interne shell sau programe specifice pot face validari de tip al variabilelor:

```
$ a="some string"
$ expr $a + 1
expr: syntax error
$ b=2
$ expr $b + 1
3
$
```

Comportamentul de mai sus este indus de comanda *expr* care asteapta valori de tip numeric.

1.3 Citirea variabilelor de la tastatura

Variabilele pot fi citite interactiv folosind comanda interna shell *read*, ca in exemplul de mai jos:

```
$ read favorite_movie
Star Trek
$ echo $favorite_movie
Star Trek
$
```

Comanda *read* poate fi folosita ca o metoda rapida de tokenizare a datelor de intrare ca in exemplul de mai jos:

```
$ read a b c
1 2 3
$ echo $a
1
$ echo $b
2
$ echo $c
3
$
```

In procesul de tokenizare, comanda *read* se foloseste de numarul de variabile primite ca parametru si asigneaza tokenii identificati in datele de intrare pe rand fiecarei variabile. Cand numarul tokenilor depaseste numarul variabilelor din linie de comanda, ultimei variabile i se asigneaza restul datelor de intrare pana la Enter, ca mai jos:

```
$ read a b c
1 2 3 4 5 6
$ echo $a
1
$ echo $b
2
$ echo $c
3 4 5 6
$
```

Ce se intampla daca sunt mai multe variabile decat tokeni in datele de intrare?

1.4 Valori implicite ale variabilelor

Variabilele pot avea si valori implicite, care sunt folosite atunci cand variabila nu este setata. Pentru acest lucru se foloseste o notatie speciala implicand acolade {}. Reluand exemplul de mai sus al variabilei *myname*, i se poate asocia o valoare implicita astfel:

```
$ echo ${mylongname:- John Doe}
John Doe
$ echo $mylongname

$
```

Dupa cum se poate observa, variabila *mylongname* nu este setata, iar expresia folosind acolade permite definirea unei valori implicite. Subsecvent, daca variabila este fie initializata, fie citita de la tastatura, valoarea ei implicita este ignorata:

```
$ read mylongname
John J. Doe
$ echo $mylongname
John J. Doe
$ echo ${mylongname:- John Doe}
John J. Doe
$
```

1.5 Variabile de mediu

Variabilele de mediu sunt folosite atât de shell cât și de unele comenzi de sistem. Prin convenție sunt scrise cu litere mari (vezi `PATH` din laboratorul trecut) și sunt manipulate în același mod ca variabilele simple. Variabilele de mediu sunt exportate de catre shell cu comanda *export* si, asa cum se va vedea mai tarziu, sunt trimise ca parametru comenzilor lansate in executie.

2 Liste de comenzi

Asa cum s-a mentionat la curs, o lista de comenzi care se executa secvential poate fi tiparita pe o singura linie la promptul shell de maniera urmatoare:

- `cmd1; cmd2; cmd3; samd` – se execută mai intai cmd1, dupa ce se termina cmd1 se executa cmd2, samd

Iata un exemplu concret:

```
$ echo -n "prima lista de comenzi a utilizatorului "; whoami; ls -l
```

In comanda de mai sus (o lista de comenzi este pana la urma si ea o comanda), *echo* $-n$ suprima afisarea caracterului *newline*.

2.1 Expresii logice

Toate shell-urile POSIX oferă acces la expresii logice pentru a procesa și lega diferite comenzi. Expresiile logice primare de tip `și` și `sau` au aceiași sintaxă ca în limbajul C: `&&`, respectiv `||`.

Comenzile efectuate în shell se execută cu succes sau nu și întorc o valoare pentru a semnala acest lucuru. Valoarea întoarsă și semnificația ei este documentată în manual. În general o valoare nulă semnifică succes și una nenulă eșec. Din această cauză **modul de operare al expresilor logice este pe dos**:

- `cmd1 && cmd2` – execută cmd2 doar dacă ieșirea lui cmd1 este zero
- `cmd1 || cmd2` – execută cmd2 doar dacă ieșirea lui cmd1 este nenulă

Valorea întoarsă de ultima comandă executată este salvată în variabila `?`. În următorul exemplu, întâi creăm un fișier nou `animals.txt` în care punem cuvântul `cat`, după care căutăm pe rând cuvintele `cat` și `dog` în acest fișier. Prima dată căutarea se întoarce cu succes (zero), iar a doua oară fără nici un rezultat deci cu valoare nenulă (unu).

```
$ echo cat > animals.txt
$ grep cat animals.txt
cat
$ echo $?
0
$ grep dog animals.txt
$ echo $?
1
```

În funcție de ieșire se iau diferite decizii. De exemplu, fie fișierul `agenda.txt` în care se găsesc datele de contact a diferitor persoane. Pentru a căuta dacă o anumită persoană există în agendă se poate folosi comanda `grep(1)`

```
$ echo Ana > agenda.txt
$ grep Ana agenda.txt
ana
$ grep Mara agenda.txt
$
```

Mai sus se constată că `ana` există în agendă, dar `mara` nu. La finalul execuției comenzii `grep(1)` aceasta iese fie cu valoarea 0, dacă s-au găsit una sau mai multe intrări, sau 1 dacă nu s-a găsit nimic. Pentru a obține un verdict mai prietenos putem folosi expresia logică `sau`

```
$ echo Ana > agenda.txt
$ grep Ana agenda.txt || echo "Persoana nu exista!"
Ana
$ grep Mara agenda.txt || echo "Persoana nu exista!"
Persoana nu exista!
```

Dacă nu ne amintim dacă am trecut-o pe Ana-Maria drept Ana sau Maria în agendă, putem de asemenea folosi expresii logice pentru a căuta după Maria în caz că Ana un există:

```
$ echo Maria > agenda.txt
$ grep Ana agenda.txt || grep Maria agenda.txt
Maria
```

3 Compilarea programelor C

3.1 Editarea ad-hoc a fisierelor

Asa cum am vazut shell-ul ofera varianta de a edita texte ad-hoc, fara a folosi un editor. De pilda, vom folosi comanda `cat(1)` pentru a scrie programe scurte C pe care să le compilăm și executăm. Când comanda `cat(1)` primește ca argument caracterul `-` (sau atunci cand nu primeste nici un fisier ca parametru), atunci comanda așteaptă date de la tastatură.

```
$ cat -
$ cat -
echo
echo
stop doing that
stop doing that
press ctrl-d
press ctrl-d
```

`cat(1)` continuă să citească date până când utilizatorul apasă simultan tastele `Ctrl` și `d`, pe scurt `Ctrl-d`. In Unix, `Ctrl-d` este caracterul EOF (End Of File).

Implicit, comanda `cat(1)` scrie la standard output caracterele pe care le-a citit, de aceea in exemplul de mai sus se repetă datele de intrare pe ecran. Pentru a scrie într-un fișier datele de la intrare putem să folosim operatorul de redirecționare `>`. Astfel avem toate ingredientele pentru a scrie primul nostru program C:

```
$ cat - > hello.c
#include <stdio.h>

int main()
{
printf("Hello, World!\n");
return 0;
}
```

după ultima acoladă } apăsați `Ctrl-d` pentru a încheia șirul de date de intrare (practic scrieti EOF in hello.c) Verificăm dacă noul fișier conține într-adevăr ce am introdus de la tastatură tot cu comanda `cat(1)`:

```
$ cat hello.c
#include <stdio.h>

int main()
{
printf("Hello, World!\n");
return 0;
}
```

Shell-urile ofera o varietate adesea luxurianta de a executa acelasi task cu diferite comenzi sau in diferite combinatii ale acelorasi comenzi. De pilda, fisierul `hello.c` poate fi editat ad-hoc asa cum am invatat la curs si laborator folosind caracterul de redirectare << si un delimitator specific:

```
$ cat << END > hello.c
> #include <stdio.h>
>
> int main()
> {
> printf("Hello, World!\n");
> return 0;
> }
> END
$ cat hello.c
#include <stdio.h>

int main()
{
printf("Hello, World!\n");
return 0;
}
$
```

Iata inca o varianta de a crea acest fisier sursa C:

```
$ cat < ‘tty‘ > hello.c
#include <stdio.h>

int main()
{
printf("Hello, World!\n");
return 0;
}
$ cat hello.c
#include <stdio.h>

int main()
{
printf("Hello, World!\n");
return 0;
}
$
```

In secventa de comenzi de mai sus, prima comanda *cat* este instruita de shell sa citeasca date de la o sursa reprezentata de rezultatul comenzii *tty* si sa scrie aceste date in fisierul `hello.c`. Ori de cate ori o comanda este incadrata in

backquotes ‘...‘ , shell-ul executa comanda respectiva si inlocuieste tot ceea ce se afla intre backquotes cu rezultatul executiei comenzii. In exemplul de mai sus, daca presupunem ca rezultatul comenzii *tty* este */dev/pts/1*, terminalul pe care lucrati, efectul net al comenzii de mai sus este echivalent cu comanda:

```
$ cat < /dev/pts/1 > hello.c
```

care nu este altceva decat a patra varianta de a edita ad-hoc un fisier sursa C folosind comanda *cat*. Observati ca atunci cand redirectati intrarea standard a comenzii *cat* catre terminalul pe care lucrati, shell-ul nu mai intervine la introducerea datelor, fapt semnalat de lipsa promptului de continuare > pe care l-am vazut mai sus cand am folosit redirectarea de tip <<. Pentru a afla exact de unde s-au citit datele in cazul vostru, folositi comanda *tty* pentru a afla terminalul pe care lucrati si implicit cum a aratat de fapt comanda

```
$ cat < ‘tty‘ > hello.c
```

pe care ati folosit-o in propriul terminal. In fine, ce se intampla daca folositi urmatoarea comanda care in esenta foloseste terminalul de lucru drept prim parametru al comenzii *cat*?

```
$ cat ‘tty‘ > hello.c
```

Cum va explicati ceea ce se intampla?

La finalul acestei subsectiuni, n-ar trebui sa uitam ca de fapt principala utilitate a comenzii *cat* este concatenarea de fisiere si afisarea rezultatului la standard output (ecran). De pilda, daca vrem sa inseram un comentariu la inceputul fisierului C `hello.c` putem recurge la urmatoarea comanda:

```
$ cat hello.c
#include <stdio.h>

int main()
{
printf("Hello, World!\n");
return 0;
}
$ cat - hello.c > new_hello.c
/* this is my first C program */
$ cat new_hello.c
/* this is my first C program */
#include <stdio.h>

int main()
{
printf("Hello, World!\n");
```

```
return 0;
}
$ mv new_hello.c hello.c
```

Obs: ultima comanda din exemplul de mai sus, *mv*, *muta* fisierul sursa, primul parametru al comenzii, in cel de-al doilea fisier furnizat ca parametru. In fapt, are loc doar redenumirea fisierului sursa C, NU SI COPIEREA SA ! Pentru a copia noul fisier creat in vechiul fisier puteti folosi comanda *cp* ca mai jos, cu observatia ca acum se vor copia efectiv datele si vor fi suprascrise datele fisierului `hello.c` iar pe disc vor exista doua copii ale aceluiasi program C:

```
$ cp new_hello.c hello.c
$ cat hello.c
/* this is my first C program */
#include <stdio.h>

int main()
{
printf("Hello, World!\n");
return 0;
}
```

Pentru a verifica efectele celor doua comenzi, *mv si cp*, folositi comanda *ls* pentru a verifica continutul directorului curent dupa executia fiecarei comenzi.

3.2 Compilarea si executarea unui program C

În sistemele de operare de tip UNIX, compilatorul de C este invocat prin comanda `cc(1)` (scurt de la *C compiler*). Acesta așteaptă ca argumente fișierele sursă și, opțional, numele executabilului rezultat în urma compilării. In Linux, compilatorul uzual de C este varianta *GNU*, iar compilatorul se numeste *gcc*.

```
$ cc hello.c -o hello
```

În comanda de sus compilatorul primește fișierul C `hello.c` și numele executabilului `hello` transmis prin opțiunea `-o` cu *o* de la *ouptut*. **Atenție**, dacă omiteți specificarea unui nume pentru executabil acesta va fi implicit denumit `a.out` din motive istorice.

Pentru a executa binarul rezultat se folosește comanda:

```
$ ./hello
Hello, World!
```

unde `hello` este numele executabilului, iar `./` spune shell-ului să nu caute executabilul în `$PATH` pentru că se află în directorul curent.

Formatul cel mai general al functiei main contine in semnatura functiei numarul argumentelor din linia de comanda (*argc*), un vector de stringuri care contine argumentele propriu-zise din linia de comanda (*argv*), si respectiv mediul de executie reprezentat ca un vector de stringuri (*envp*). Fiecare element al

vectorului care reprezinta mediul de executie are forma pe care am discutat-o la curs si laborator *nume=variabila*. Aceasta este modalitatea prin care shell-ul comunica mediul de executie comenzilor pe care le lanseaza si care astfel pot folosi valorile variabilelor de mediu pentru a isi adapta comportamentul la mediul de executie.

Compilati si executati similar cu programul `hello.c` de mai sus programul `printenv.c` de mai jos, care tipareste prima variabila de mediu din lista.

```
#include <stdio.h>

int main(int argc, char *argv[], char *envp[])
{
printf("Prima variabila de mediu este %s\n", envp[0]);
exit(0);
}
```

Obs: Asa cum am mentionat anterior, fiecare comanda (program executabil) Unix intoarce un cod de retur folosit asa cum am vazut in comenzi conditionale, de pilda. Conventia Unix este ca orice program care intoarce cod de retur 0 s-a terminat cu succes in vreme ce un cod nenul (uzual pozitiv) desemneaza o conditie de eroare. Din acest motiv, este o buna practica de programare sa incheiati programele folosind apelul sistem *exit* care este responsabil pentru a comunica acest cod de retur care e returnat de catre shell prin evaluarea variabilei *?*.

4 Sarcini de laborator

1. Executați toate comenzile prezentate în acest laborator.
2. Folositi comanda *read* in cadrul unei liste de comenzi care citeste de la tastatura trei intregi si afiseaza suma lor.
3. Comanda *whoami* returneaza ID-ul de utilizator (echivalent cu *id -un*). Cum puteti folosi aceasta comanda pentru a seta valoarea implicita a unei variabile numita *myusername* ?
4. Compilați și executați programul `hello` doar dacă compilarea a decurs cu succes. Faceți asta într-o singură comandă folosind expresii logice.
5. Modificați programul `hello` să citească un nume cu `scanf` (ex. Alex) pe care să-l salute pe urmă cu ajutorul funcției `printf` (ex. "Hello, Alex!"). Compilați și executați.
6. Modificați programul `printenv` de mai sus a.i. sa tipareasca toate variabilele de mediu (**N.B.** vectorul *envp* se termina cu NULL). Compilați și executați.

7. Creați un director `bin` în care copiați executabilul `hello`. Adăugați directorul `bin` (**folosind calea absolută**) în variabila `$PATH` și arătați că puteți executa simplu cu comanda `hello` fără a avea nevoie de prefixul `./`.